(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 788 055 A2

(12) **EUROPEAN PATENT APPLICATION**

(43) Date of publication:
06.08.1997 Bulletin 1997/32

(51) Int Cl.^{6}: **G06F 13/16**, G06F 13/12

(21) Application number: 97300431.0

(22) Date of filing: 23.01.1997

(84) Designated Contracting States:
DE FR GB

(30) Priority: 05.02.1996 US 597092

(71) Applicant: INTERNATIONAL BUSINESS MACHINES CORPORATION
Armonk, NY 10504 (US)

(72) Inventors:
- Hodges, Paul
San Jose, California 95120 (US)
- Hurley, Michael Garwood
Cupertino, California 95014 (US)
- Ouchi, Norman Kenneth
San Jose, California 95120 (US)
- Shih, Mien
Saratoga, California 95070 (US)

(74) Representative: Ling, Christopher John
IBM United Kingdom Limited,
Intellectual Property Department,
Hursley Park
Winchester, Hampshire SO21 2JN (GB)

(54) **Multipath i/o storage systems with multipath i/o request mechanisms**

(57) A data processing system (300, 800, 1000) having multiple independent paths for communication between a host (310, 810, 1010) and a plurality of storage devices (340, 840, 1040) where each path has its own queue (326, 828) for servicing requests generated by the host for accessing the storage devices. Each request is assigned a unique sequential ID before it is stored, along with its unique ID, in all the queues. Each storage device has a "mailbox" register (500) where the ID and the status (520) of the latest request being carried out is stored. Queues are serviced and their status updated based on the content of the mailbox in each storage device. The combination of assigning a unique task ID to each request and a "mailbox" register in each storage device allows the queue in each path to be completely out of sync with each of the queues in the other paths without causing data integrity problems, duplication of requests at the device level, or a need for complex locking schemes to keep the queues in sync with each other.

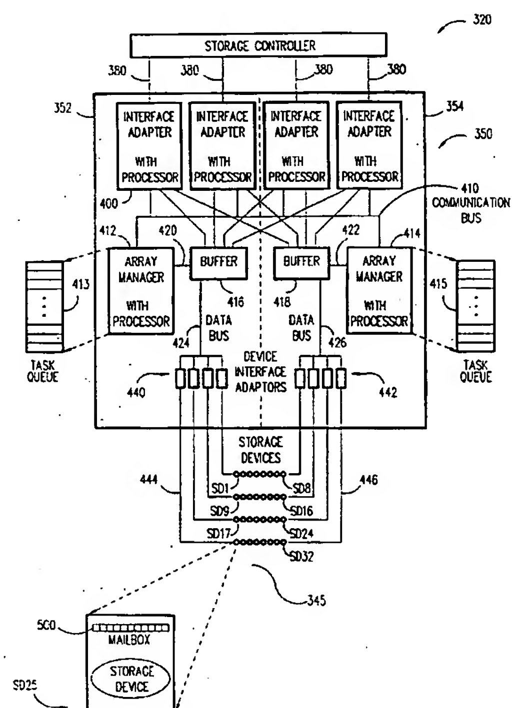


FIG. 5

EP 0 788 055 A2

Printed by Jouve, 75001 PARIS (FR)

Description

Technical Field

This invention relates to a data processing system having multiple independent paths for communication between multiple independent storage controllers and storage devices. Specifically, this invention relates to a method and means for efficient management of the queues in a multiple independent path storage subsystem where the requests for accessing the storage devices can be carried out without the need for the queues to be in sync with each other.

Background Art

Data processing systems (systems) with multiple input/output (I/p) storage subsystems generally have multiple independent communication paths between the processor and each storage device in the system. A typical data processing system 100 having such a feature is shown in Fig. 1. Host 110 generally comprises an application program 112, operating system 114, and an I/O supervisor 116 where the I/O supervisor further includes a host task queue 117 for managing the requests issued by the host. Host 110 further comprises a plurality of I/O channels 118 for communication with storage controller 120. Storage controller 120 generally comprises a plurality of I/O ports 122 for communication with host 110, a shared cache 124 for high performance, and a plurality of controller paths 130 for accessing storage devices 140. Storage controller 120 and storage devices 140 are generally referred to as a storage subsystem.

In general, if an I/O request issued by host 110 cannot be satisfied by information already stored in cache 124, storage controller 120 will access the appropriate storage device via one of the available controller paths 130 to carry out the I/O request. The data processing system of Fig. 1 in general provides high availability due to redundancy of the storage subsystem, multiple I/O channels, multiple controller paths in the storage controller, and multiple communication links between the storage controller and the storage devices.

In this type of system, a typical queuing of an I/O request issued by host 110 is carried out as follows: I/O request is initiated by application program 112 and passed to I/O supervisor 116. I/O supervisor 116 receives and adds the request to host task queue 117 which is maintained by the I/O supervisor. When one of the communication links 150 becomes available, I/O supervisor 116 initiates an I/O process for the first request in the queue 117 for which the corresponding device is available.

In this type of system, since host task queue 117 is the only queue of I/O operation available in the system, all operations are initiated at host 110 and only one operation can be active for any device. Furthermore, that operation must be reported as complete before another operation for that device can be initiated by I/O supervisor 116.

In this type of architecture, failure of one of the channels 118, communication links 150, device communication links 160 or controller paths 130 does not prevent access to storage devices, but a failure that affects host task queue 117 will cause requests in host task queue 117 to fail and is likely to cause one or more programs to abort. Recovery from failure may involve reexecuting the program on the same host or on a different host system.

However, in a data system where the storage controller has a cache, it is desirable that operations between cache 124 and storage devices 140 be performed concurrently with operations between cache 124 and host (also referred to as central processing unit (CPU)) 110. For example, a request to write data to storage device 142 issued by host 110 can be considered complete by host 110 when the data has been transferred to cache 124. After that, subsequent operations for device 142 can be executed from cache 124 while the updated data is written into storage device 142.

An example of a data system where the storage controller has a cache is shown in Fig. 2. Fig. 2 represents an IBM System/390 (host 110) in communication with IBM storage controller 3990 (storage controller 120) which controls the operation of IBM magnetic storage devices 3390 (storage devices 140). In this system, there are four communication paths (data paths) between storage controller 120 and storage devices 140. Each data path comprises a controller path 130 and a device communication link 160 and is available for carrying instructions to perform operations on any of the storage devices 140.

In order to provide high availability, storage controller 120 is generally divided into two storage sub-controllers 132 and 134. Storage sub-controller 132 comprises a controller task queue 126 and a plurality of controller paths 130 (two controller paths shown in Fig. 2). Storage sub-controller 134 comprises a controller task queue 128 and a plurality of controller paths 130 (two controller paths shown in Fig. 2). Task queues 126 and 128 are replicas of each other. Furthermore, the two storage sub-controllers are designed to be at different power boundaries to improve overall system reliability. Therefore, if one of the storage sub-controllers fails, the other storage sub-controller would continue to execute requests from its task queue thus providing continuous performance although at a reduced level.

Considering that each request for access to a storage device issued by host 110 may contain several subcommands which any of the sub-commands may be initiated by one sub-controller and completed by another, the management of the task queues in storage controller 120 becomes an extremely serious and critical issue. For example, an I/O request usually contains two subcommands, 1) a preparatory state command of "seek/locate" and 2) a time dependent data transfer command

known as "read/write". In the case of this type of an I/O request, either one of the storage sub-controllers may be available to carry on any of the sub-commands at any given time and may indeed service the next operation on the queue.

Therefore, each storage sub-controller has to keep track of the state of the I/O request for each device to ensure that a read/write command is associated with the correct I/O request before it is carried out by either one of the sub-controllers. That is, there must be a mechanism by which acknowledgment of completion of the seek/locate sub-command is received by both sub-controllers. For example, execution of the read/write sub-command by either one of the sub-controllers might be delayed until both sub-controllers have received acknowledgment from the storage device. However, this approach would make access to storage devices extremely slow, and would also compromise the independence of the storage sub-controllers.

One way to address this problem is to ensure that the storage sub-controllers operate independently to ensure high availability yet communicate very closely with each other in processing device access requests to improve performance. Such an architecture is shown in Fig. 2 where a request for access to a storage device is replicated by each storage sub-controller and sent by each sub-controller to the device. In this architecture, the first storage sub-controller acquiring the device and establishing a communication path transmits the seek/locate command to the device. The communication path is then disconnected from the device and the storage device begins executing the seek/locate sub-command. Once the operation is complete and the device is ready for data transfer, the device raises a flag or interrupt to inform the storage controller. Once, the device raises a flag or interrupt, the first available sub-controller sensing the interrupt from the device acquires the device and completes the data transfer from the device to the shared cache 124 in storage controller 120.

Therefore, in this architecture, the storage sub-controllers operate independently to provide high availability yet cooperate very closely to ensure high performance. However, since the communication paths through each storage sub-controller are asynchronous, which means requests may be delayed through one communication path compared to another path, this can very well result in a storage sub-controller executing a request or a sub-command which has already been completed by the other storage sub-controller. This could easily lead to wasted operations which lowers overall storage subsystem performance and at the same time could cause data integrity problems.

A high speed message passing architecture may be utilized in the system of Fig. 2 between the storage sub-controllers to inform a sub-controller of the operations executed by the other sub-controller. But even sending messages from one sub-controller to another may be delayed leading to duplication of requests and data integrity problems.

Furthermore, since a plurality of communication paths exist between storage controller 120 and storage devices 140 and any one of the communication paths that are available may service the next I/O request from either one of the task queues 126 and 128, the two copies of the queues must be kept identical at all times to prevent executing the same I/O request twice. This means a sophisticated and complex locking scheme must be used to ensure that a sub-controller ready for work has exclusive access to both copies of the queues and that both copies of the queue are updated before the lock is released.

Therefore, while the two storage sub-controllers of storage controller 120 are intentionally independent with respect to hardware failures, the communication between the two sub-controllers is very critical to ensure queue integrity. The dependency on complex and elaborate locking schemes and high speed communication between the sub-controllers of the storage controller results in a slowing down of system performance and can lead to performance bottlenecks.

One way to eliminate such a bottleneck and dependency on complex locking schemes between the queues is to eliminate storage controller 120 and move the necessary function to each storage device. Such an architecture is shown in Fig. 3 in which a plurality of processors 210 and 212 are in communication with multiple storage devices 220 via a small computer system interface (SCSI) bus 218. In this architecture, each storage device 220 contains a device controller 224 and a storage medium 222. Each SCSI device controller 224 further comprises its own device task queue 226.

Maintaining a queue of work for a storage device at the device level itself is efficient from the queuing standpoint, but such a device requires additional hardware and software, and must have a sophisticated device controller. Although devices having sophisticated device controllers are available (SCSI devices), there is also a desire to have a simple storage device having no device controller that can be directly connected to a storage controller or a RAID controller or a network-attached data server without the problems associated with managing multiple queues at the storage controller.

Therefore, in a storage subsystem having a plurality of storage devices in communication with a storage controller via a plurality of independent I/O communication (data) paths, there is a need for a method and means for coordinating I/O requests maintained in multiple task queues and cancelling completed operations among the independent communication paths without the need for keeping the task queues in the storage sub-controller in complete synchronization with each other, without the need to provide continuous status of each storage device and request between the storage sub-controllers, and without the need for high speed communications between the sub-controllers in an attempt to keep the queues in sync with each other.

Also, in a data processing system having a plurality of independent storage controllers in communication with storage devices via a plurality of independent I/O communication (data) paths, there is a need for a method and means for processing I/O requests maintained in multiple task queues in the controllers and cancelling completed operations among the independent communication paths without the need for keeping the task queues in the controller in complete synchronization with each other, without the need to provide continuous status of each storage device and request between the storage controllers, and without the need for high speed communications between the storage controllers in an attempt to keep the queues in sync with each other.

Disclosure of the Invention

In accordance with the present invention, in a data processing system having multiple independent I/O paths between a storage controller and storage devices wherein the storage controller comprises a plurality of storage sub-controllers and wherein each storage sub-controller comprises a task queue (queue), a method and means are disclosed wherein the status of each queue and the I/O request (job) completed or in progress are communicated to the other queues via a register that is maintained at each storage device. This is done by:

(1) assigning a unique identifier to each I/O request before the request is stored in the queues. The unique identifier for each I/O request stored in the queues will ensure that an I/O request is removed from all copies of the queues when that request is completed through any one of the storage sub-controllers;

(2) maintaining a register known as a "mailbox" in each storage device which can be read and/or written to by any sub-controller that establishes a communication path with that device. The mailbox contains the unique identifier and the status of the last I/O request executed or the I/O request currently being executed by that device; and

(3) allowing any of the storage sub-controllers to read the content of the mailbox and take the appropriate action depending on the content of the mailbox.

The appropriate action taken by a storage sub-controller reading the mailbox of a device may result in changing the status of the storage device and updating the mailbox content to reflect the change in the status. The sub-controller reading the mailbox may also update its copy of the queue on the basis of what it just read.

Therefore, based on the disclosed invention, the copies of the queues are not necessarily in sync with each other. Indeed, the queues could be completely out of sync with each other without causing any duplication of work, data integrity problems, or slowing down the processing of the I/O requests.

For example, if a request to read data from storage device A is issued by the host, the request is assigned a unique label x by the storage controller before the request is stored in the task queues. Once device A becomes available, the request for access to device A is dispatched through all available communication paths. If path 1 is the first to acquire device A, it issues the seek/locate command to prepare the device for data transfer. Device A then stores the unique label x in its mailbox and sets the state of I/O operation to show that device A has been issued the command to prepare itself for data transfer. Path 1 then releases device A and carries out other operations while device A is preparing itself for data transfer. Assuming that path 2 is the first path to sense that device A is ready for data transfer, path 2 reads device A mailbox to find out that the request with unique label x is in progress, associates the data transfer with the request labelled x in its task queue, and then executes the data transfer. Once the data transfer is completed, the state of that specific I/O request at device A is set to complete to reflect that the request associated with the unique label x has been carried out. Assuming that path 1 now attempts to read device A, once it reads device A's mailbox it will find out that the request associated with unique label x has already been completed through another path. So, path 1 deletes the request associated with unique label x from its own task queue and then releases device A.

Using a unique label for each I/O request stored in the task queues in conjunction with a mailbox at each storage device prevents the same request from being executed more than once by the storage sub-controllers. At the same time, it allows the task queues to be updated asynchronously without affecting the performance of a storage system. Furthermore, using a unique label for each I/O request in conjunction with a mailbox at each storage device eliminates the need for having sophisticated and complex lock step schemes to ensure that the task queue in each storage sub-controller is in sync with the task queues in the other storage sub-controllers. Using the present invention also eliminates the need for providing high speed direct communication between storage sub-controllers to quickly and continuously communicate the queues' status because task queues need not be in sync with each other.

Brief Description of the Drawings

The invention will now be described, by way of example only, with reference to the accompanying drawings, in which:

Fig. 1 is a depiction of a prior art data processing system having a host in communication with a stor-

age subsystem;

Fig. 2 is a depiction of another prior art data processing system having a plurality of task queues at the storage controller;

Fig. 3 is a depiction of another prior art data processing system having a task queue in each device;

Fig. 4 is a depiction of the preferred embodiment of the present invention;

Fig. 5 is a detailed schematic diagram of the array controller shown in Fig. 4;

Fig. 6 is a depiction of a mailbox stored in each storage device;

Fig. 7 is a flowchart of I/O requests handling in the preferred embodiment of the present invention;

Fig. 8 is a depiction of an alternative embodiment of the present invention;

Fig. 9 is a depiction of another alternative embodiment of the present invention; and,

Fig. 10 is a depiction of another alternative embodiment of the present invention.

Detailed Description of the Invention

With respect to Fig. 4, there is shown the preferred embodiment of the present invention. Data processing system 300, comprises a host 310 in communication with storage controller 320, where the storage controller is in communication with storage array 340. The storage array 340 further comprises an array controller 350 which controls access to storage devices 345. In this embodiment, the storage devices that are normally attached to storage controller 320 are replaced with storage array 340. Storage array 340 then can be treated as a logical IBM 3390 storage device. That is, storage array 340 can allow attachment of standard storage devices to storage controller 320 by providing emulation of previously existing storage devices such as the IBM 3390 storage subsystem. Furthermore, array controller 350 in combination with storage devices 345 can provide extremely high data availability via architectures such as reliable arrays of inexpensive disks (RAID).

Referring back to Fig. 4, host 310 comprises an application program 312, operating system 314, an I/O supervisor 316, and a plurality of I/O channels 318 for communication with storage controller 320. Storage controller 320 generally comprises a plurality of I/O ports 322 for communication with host 310, a shared cache 324 for high performance, and two storage sub-controllers 331 and 332 for accessing storage array 340. Storage sub-controller 331 comprises a controller task queue 326 and controller paths 333 and 334. Storage sub-controller 332 comprises a controller task queue 328 and controller paths 335 and 336. Storage controller 320 communicates with storage array 340 via a plurality of communication lines 380.

Fig. 5 shows the detailed schematic diagram of array controller 350 in communication with storage devices 345 and storage controller 320. Array controller 350 comprises two identical independently operating controller paths 352 and 354. Controller paths 352 and 354 together provide for enhanced performance and fault tolerant operation. In the preferred embodiment, array controller 350 is in communication with storage controller 320 over a plurality of communication links 380 (four shown in this example). Array controller 350 further comprises a plurality of storage interface adapters 400 (four shown in this example, two in each controller path). Storage interface adapters 400 receive commands (instructions, requests) for access to storage devices 345 from storage controller 320 and initially determine if a request issued by storage controller 320 can be satisfied from either buffer 416 or buffer 418. Buffers 416 and 418 store data received from the devices and the storage controller. If the request can be satisfied from the content of either buffer, that information is communicated back to storage controller 320 via one of the storage interface adapters 400. For each request that cannot be satisfied from the content of one of the buffers, one of the storage interface adapters generates a unique request identifier (also referred to as a label) and sends the request with its unique identifier to array managers 412 and 414. Array managers 412 and 414, in turn, add the request and its label to their respective queues 413 and 415.

Array managers 412 and 414 are generally responsible for managing data handling between storage controller 320 and storage devices 345. They further translate data format between storage devices 345 format (native format) and the emulated device format (the format as seen by storage controller 320). For example, the data might be stored on storage devices 345 as a fixed block format (native device format) . On the other hand, storage controller 320 may be set up to handle the information in a count-key-data format. Therefore, array managers 412 and 414 provide the translation between these two different types of data formats. Each array manager further maintains its own task queue. For example, array manager 412 maintains array task queue 413 and array manager 414 maintains array task queue 415. Each array manager updates its own array task queue on the basis of messages received from any one of the storage interface adapters 400 and based on the content of the mailbox in each storage device. Array managers 412 and 414 further manage dispatching device operations to storage devices 345 via device interface adapters 440 and 442, respectively.

In the preferred embodiment, each array controller path 352 and 354 includes four device interface adapters. Array managers 412 and 414 further notify device interface adapters 440 and 442, respectively, when a request to access a device or data is available for transmission. Array managers 412 and 414 are also in communication with each other via communication bus 410 through which they may notify each other of the events that may affect the operations of their respective array controller paths 352 and 354. Array managers 412 and 414 may further include RAID exclusive OR (XOR) functions to maintain parity across storage devices 345 to ensure redundant recording of data and fault tolerant operations.

As mentioned above, device interface adapters 440 and 442 provide access to devices 345 and control storage devices' operations. Device interface adapters also provide operation status to their respective array manager. Fig. 5 further shows a plurality of storage devices 345 (32 devices shown in this example and labelled SD1 through SD32) each device having a mailbox 500. In Fig. 5, storage devices 345 are arranged in four clusters, each cluster including eight devices. Devices 345 can be accessed independently by array controller paths 352 and 354 through their respective device interface adapters 440 and 442. Each device interface adapter can access four storage devices concurrently, one from each cluster.

With respect to Fig. 6, there is shown a representation of a device mailbox 500 maintained in each of the storage devices 345. In the preferred embodiment, mailbox 500 comprises an eight-byte field for storing the unique request ID 510 (task ID) and the status 520 (task status) of the latest request that has been received by a storage device from array controller 350. The unique request (task) IDs 510 are preferably, but not necessarily, sequential for ease of handling and operation. Mailbox 500 may also include information such as device interface adapter identification from which the request was received. Mailbox 500 may also include information such as device allegiance.

With reference to Fig. 7, there is shown a flowchart of device access and queues management operation carried out by array managers 412 and 414 in the preferred embodiment of the present invention. The operation will be described with respect to array manager 412 although it is as applicable to array manager 414.

Array manager 412 initially determines whether any message with respect to queues' status has been received from array manager 414 via communication bus 410 (block 705). Array task queue 413 is updated accordingly if such a message is received (block 706) otherwise array manager 412 determines whether an interrupt from any of the devices 345 has been received (block 710). If a device interrupt from device n is received, device n is selected and its mailbox is read to determine what is the task ID (request ID) and the status of the latest request dispatched to device n (blocks 712 and 715). Assume that the content of mailbox register indicates that the task with unique sequential ID "R" (referred to simply as task "R") is in progress. Array manager 412 then updates array task queue 413 and deletes all the requests having unique sequential ID less than "R" and updates queue 413 to reflect that task "R" is in progress (block 716). Array manager 412 then issues a command to device n to complete task "R" (block 717) and at the same time updates the mailbox in device n to post "task R complete" (block 718). Array manager 412 then deselects device n and deletes task "R" from its array task queue 413 (block 720).

Referring back to block 715, if the content of mailbox register in device n indicates that the task "R" has been completed, array manager 412 then updates array task queue 413 by deleting task "R" and all the requests having unique sequential ID less than "R" (block 725). Array manager 412 then checks array task queue 413 to determine if there is any other tasks in the queue for device n (block 726); if there is none, the operation returns to block 700; if there is a task "S" for device n, array manager 412 issues commands to initiate task "S" execution (block 727), the mailbox is updated to reflect that task "S" is in progress (block 728), and device n is deselected and the status of task "S" in queue 413 is updated (block 729).

Returning back to block 710, if array manager 412 does not receive an interrupt from any device, it then checks queue 413 to determine what is the next task that should be carried out and for what device (block 740). Assume that the next task to be carried out by array manager 412 according to its queue 413 is task "T" for device n. Array manager 412 then issues commands to select device n and read its mailbox to determine the ID and the status of the latest task that was issued to device n (block 742). If the task ID in device n mailbox is less than "T" (block 745) and its status is in progress (block 770), then an error has occurred and the error is communicated back to array manager 412 (block 775). If the task ID in device n mailbox is less than "T" (block 745) and its status is complete, array manager issues commands to initiate task "T", updates the mailbox for device n to show that task "T" is in progress, and then deselects and updates the status of task "T" in array task queue 413 (blocks 771, 772, and 773).

If the task ID in device n mailbox is "U" which is greater than "T" (block 745) and the status of task "U" is complete, array manager 412 deletes tasks "T" and "U" from queue 413 (block 755) and determines if there is any other task in queue 413 for device n (block 756); if there is none, the operation returns to block 700; if there is a task "V" in queue 413 for device n, array manager 412 issues commands to initiate task "V" execution (block 757), the mailbox is updated to reflect that task "V" is in progress (block 758), and device n is deselected and the status of task "V" in queue 413 is updated (block 759).

If the task ID in device n mailbox is "U" which is

greater than "T" (block 745) and the status of task "U" is "in progress", array manager 412 then updates array task queue 413 and deletes all the tasks (requests) having task IDs less than "U" and updates queue 413 to reflect that task "U" is in progress (block 751). Array manager 412 then issues a command to device n to complete task "U" (block 752) and at the same time updates the mailbox in device n to post "task U complete" (block 753). Array manager 412 then deselects device n and deletes task "U" from its array task queue 413 (block 754).

Now referring generally to Figs. 4 through 7, the operation of task queues 413 and 415 and the use of the mailbox will be further explained with the following example. Assume task queue 413 contains unique sequentially numbered requests 25 and 26 for storage device 8 (SD8). The request 25 is at the top of the queue (which means it is the next operation to be carried out). Array manager 412 then issues a command through device interface adapter 440 to see whether storage device 8 is available or is busy. If storage device 8 is busy, that information is communicated back to array manager 412 which may then decide to carry out requests for other devices before making another attempt for communication with storage device 8. On the other hand, if storage device 8 is available, array manager 412 selects device 8 and immediately reads mailbox 500 in storage device 8 to determine what is the status of the last request that was carried out by storage device 8. If mailbox 500 shows that the last request that was completed was request number 24, then array manager 412 issues the seek/locate command to storage device 8 for request 25. It then updates the mailbox to show that request 25 is in progress. This is done by setting task status 520 at p (where p stands for "in progress"). Array manager 412 then deselects storage device 8, updates queue 413 to show that request 25 is in progress. Array manager 412 may also send a message to array manager 414 via communication bus 410 informing array manager 414 that request 25 is in progress. Array manager 412 then may return to an idle state or establish communication with the other devices.

On the other hand, if array manager 412 selects storage device 8 and after reading mailbox 500 finds out that the seek/locate command for request 25 is already in progress, array manager 412 updates queue 413 to show that request 25 is in progress, it then issues commands to device 8 to complete request 25, updates the mailbox to reflect the change in the status, and deselects storage device 8 and deletes request 25 from queue 413.

Alternatively, if array manager 412 establishes a communication link with device 8 in order to execute request 25 and after reading device 8 mailbox finds out that the seek/locate is already in progress for request 26, array manager 412 will then conclude that request 25 has already been completed by array manager 414. Array manager 412 then deselects storage device 8 and deletes request 25 from queue 413. It also updates queue 413 to show that request 26 is in progress and then returns to an idle state.

On the other hand, if array manager 412 attempts to carry out request 25 for storage device 8 and finds out, by reading mailbox 500, that request 26 has already been completed, array manager 412 will then conclude that request 25 has already been completed and that both requests 25 and 26 have been serviced by array manager 414. Array manager 412 then deletes both requests 25 and 26 from queue 413 and then searches queue 413 to determine whether there are any other requests in queue 413 for device 8. If such a request is found, then it will attempt to carry it out. If no such request for storage device 8 is found, it deselects storage device 8 and it searches queue 413 for requests for other devices.

Therefore, based on the detailed description and the examples provided, it can readily be appreciated that servicing a request having a plurality of sub-commands (two, three or more sub-commands) may be initiated by one array manager using the information in its own respective array task queue and the same request may be further carried out or completed by another array manager. It can also readily be appreciated that several requests which are present in both the array task queues may be serviced by only one of the array managers without the other array manager ever having time to participate in such servicing.

However, using a mailbox register at each storage device provides an easy and efficient way to provide the status of each request to each array manager in an array controller. Thus, queues in each controller path could be completely out of sync with each other without creating any data integrity problems or a potential for duplicating the same request. Through the use of unique request labelling in conjunction with a mailbox at each register, the status of each request (complete, progress, waiting in the queues) can be communicated between the array managers without the need for a separate high speed communication link or without the need for a complex locking scheme to ensure that the array task queues are in sync at all times.

Fig. 8 shows an alternative embodiment of the present invention. Data processing system 800 comprises a host 810 in communication with storage controller 820 where the storage controller is in direct communication with storage devices 840. In this embodiment, unlike the preferred embodiment shown in Figs. 4 and 5, storage controller 820 directly controls access to devices 840. Host 810 comprises application programs 812, operating system 814, I/O supervisor 816 and a plurality of communication channels 818.

Storage controller 820 comprises a plurality of I/O ports 822 for communication with host 810, a shared cache 824, and two storage sub-controllers 831 and 832. Storage sub-controller 831 comprises a task queue 826 and controller path 829. Storage sub-controller 832

comprises a task queue 828 and controller path 830. Each controller path further comprises a processor and a memory unit. Storage sub-controllers 831 and 832 are in communication with storage devices 840 via communication channels 846 and 847. Each of the storage devices 840 comprises a mailbox which stores the unique ID and the status of the latest request carried out by each of the storage devices. The process of accessing the devices, reading the mailboxes, and managing the task queues by the storage sub-controllers are similar to the one for the preferred embodiment of the present invention. Note that in this embodiment, task queues 826 and 828 could be completely out of sync with each other without slowing down system operation, without causing data integrity problem or without causing duplication of requests.

Fig. 9 is a depiction of another alternative embodiment of the present invention similar to the embodiment shown in Fig. 8. In this alternative embodiment, data processing system 900 comprises a storage controller 920 where the storage controller comprises four storage sub-controllers 930, 931, 932, and 933. Each storage sub-controller further has its own task queue. Therefore, in this embodiment, there are four task queues in the storage controller that may service requests issued by host 910. Using the disclosed invention, the task queues could be completely out of sync with each other without creating data integrity problems, duplication of requests at the device level or complicated locking schemes for the queues.

Fig. 10 is a depiction of another alternative embodiment of the present invention where a host may communicate with storage devices via a plurality of storage controllers connected together in a network. In this alternative embodiment, data processing system 1000 comprises a host 1010, a plurality of storage controllers 1020A, 1020B, ..., and 1020N, and storage devices 1040. Each storage controller comprises a cache and may also comprise one or more storage sub-controllers where each storage sub-controller has its own task queue. Therefore, in this embodiment, a host 1010 request is sent to two or more controllers for processing. The unique task ID of the request is preferably, but not necessarily, generated by host 1010 at the same time that the request is generated and sent to the controllers. If the request cannot be processed from the information already stored in and of the caches, it will be stored in the task queues and will be processed by accessing storage devices 1040 in a manner similar to the preferred embodiment of the present invention. Using the disclosed invention, the task queues in different controllers could be completely out of sync with each other without creating data integrity problems, duplication of requests at the device level or complicated locking schemes for the queues.

A number of embodiments of the present invention have been described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and the scope of the invention. For example, the number of task queues may be any number without imposing any limitation on the invention disclosed herein. The mailbox may be expanded to keep additional information such as the history of the past n requests handled by any device and which path initiated or completed the requests. The mailbox may further include information such as aborted requests and whether the device access is limited to a specific path.

Claims

1. A data processing system (300, 800, 1000), comprising:

   a central processing unit (CPU) (310, 810, 1010);

   a controller (320, 820, 1010) in communication with said CPU, the controller having a queue (326, 826), said queue storing requests received from said CPU, each of said requests having a unique identification label; and,

   a storage device (340, 840, 1040) for servicing a request received from any one of said queues, said storage device having a mailbox (500) for storing the status and the unique identification label of the request being serviced.

2. A data processing system (300, 800, 1000) as claimed in claim 1, comprising a plurality of said controllers (320, 820, 1020), each controller having a queue (326, 826).

3. A data processing system (300, 800, 1000) as claimed in claim 1, further comprising:

   a storage subsystem in communication with said CPU, said storage subsystem comprising said controller, said controller having queues concurrently storing requests received from said CPU.

4. A data processing system (300, 800, 1000) as claimed in claim 1 in which said storage controller has a plurality of storage sub-controllers, each of said storage sub-controllers containing said queue, said queues concurrently storing said requests received from said CPU.

5. A data processing system (300, 800, 1000) as claimed in any one of claim 1 to claim 4, wherein said request comprises a plurality of sub-commands.

6. A data processing system (300, 800, 1000) as claimed in claim 5, wherein said mailbox (500) further comprises a status field (520) for storing the

status of each sub-command being executed.

7. A data processing system (300, 800, 1000) as claimed in claim 6, wherein said controller (320, 820, 1020) comprises means for reading said mailbox (500) and updating the status of the request being serviced in its queue based on the content of said mailbox.

8. A data processing system (300, 800, 1000) as claimed in claim 7, wherein each request stored in said queues comprises a plurality of sub-commands.

9. A data processing system (300, 800, 1000) as claimed in claim 8, wherein said mailbox (500) comprises a status field (520) for storing the status of each sub-command being executed.

10. A method for use in a data processing system (300, 800, 1000), comprising a host (310, 810, 1010) and a storage subsystem (320, 820, 1020) where said storage subsystem includes a storage controller and a storage device (340, 840, 1040) for processing requests received from the host and where said storage controller includes a plurality of queues (326, 826), the method of updating the status of each of said queues, comprising the steps of:

assigning a unique identification label to each request concurrently stored in said queues;

maintaining (118, 728, 753, 758, 772) a mailbox in said storage device for storing the status of each request being serviced;

reading (712, 742) said mailbox to determine the status of the last request being serviced; and,

updating (706, 716, 720, 725, 729, 751, 754, 755, 759, 773) said queues based on the content of the mailbox.

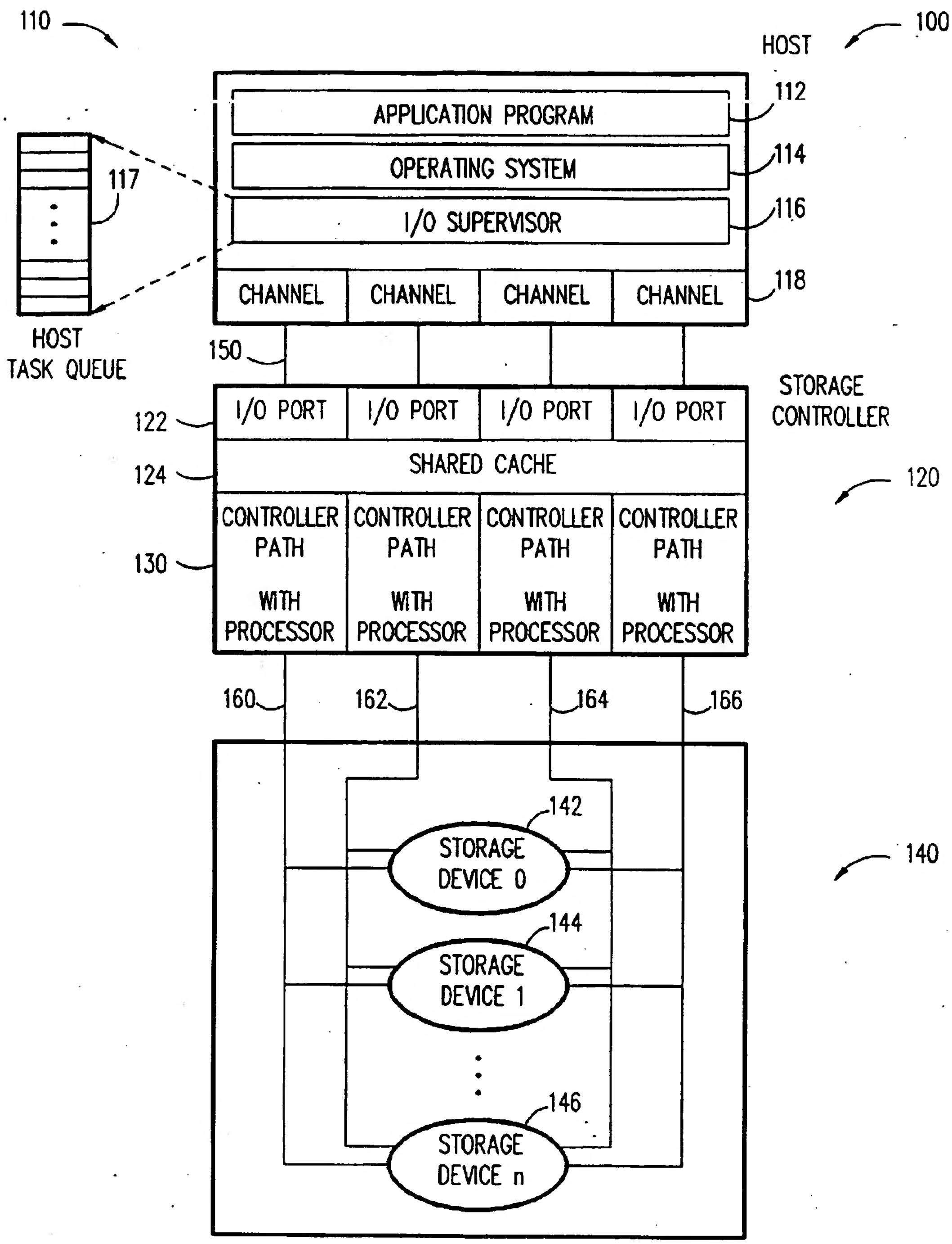

110
100
HOST
APPLICATION PROGRAM
112
OPERATING SYSTEM
114
I/O SUPERVISOR
116
117
CHANNEL
CHANNEL
CHANNEL
CHANNEL
118
HOST
TASK QUEUE
150
STORAGE
CONTROLLER
I/O PORT
I/O PORT
I/O PORT
I/O PORT
122
SHARED CACHE
124
120
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
130
160
162
164
166
142
STORAGE DEVICE 0
140
144
STORAGE DEVICE 1
146
STORAGE DEVICE n

FIG. 1

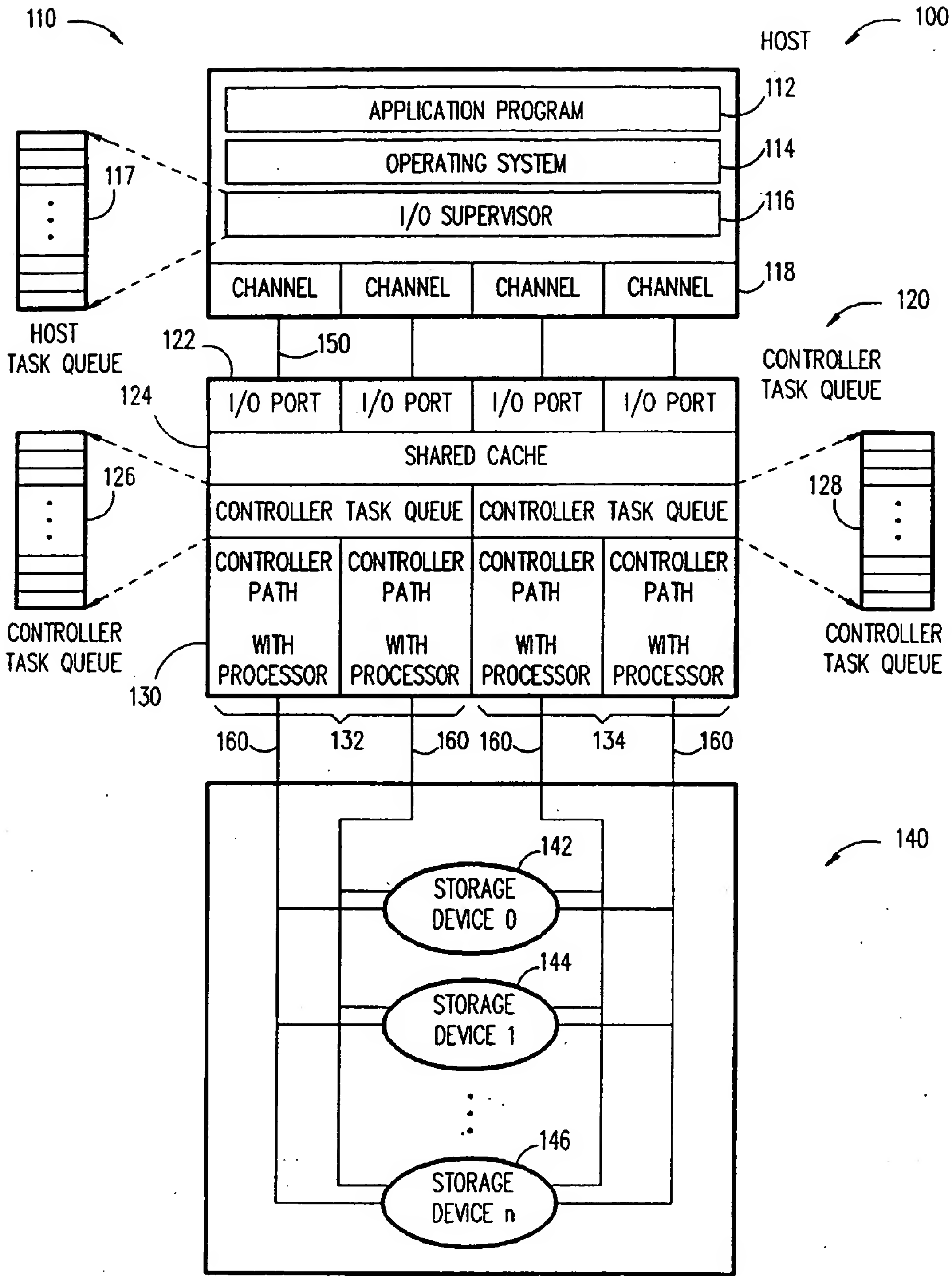

110
100
HOST
APPLICATION PROGRAM
112
OPERATING SYSTEM
114
I/O SUPERVISOR
116
117
HOST TASK QUEUE
CHANNEL
CHANNEL
CHANNEL
CHANNEL
118
120
122
150
CONTROLLER TASK QUEUE
124
I/O PORT
I/O PORT
I/O PORT
I/O PORT
SHARED CACHE
126
128
CONTROLLER TASK QUEUE
CONTROLLER TASK QUEUE
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER TASK QUEUE
CONTROLLER TASK QUEUE
130
160
132
160
160
134
160
140
142
STORAGE DEVICE 0
144
STORAGE DEVICE 1
146
STORAGE DEVICE n

FIG. 2

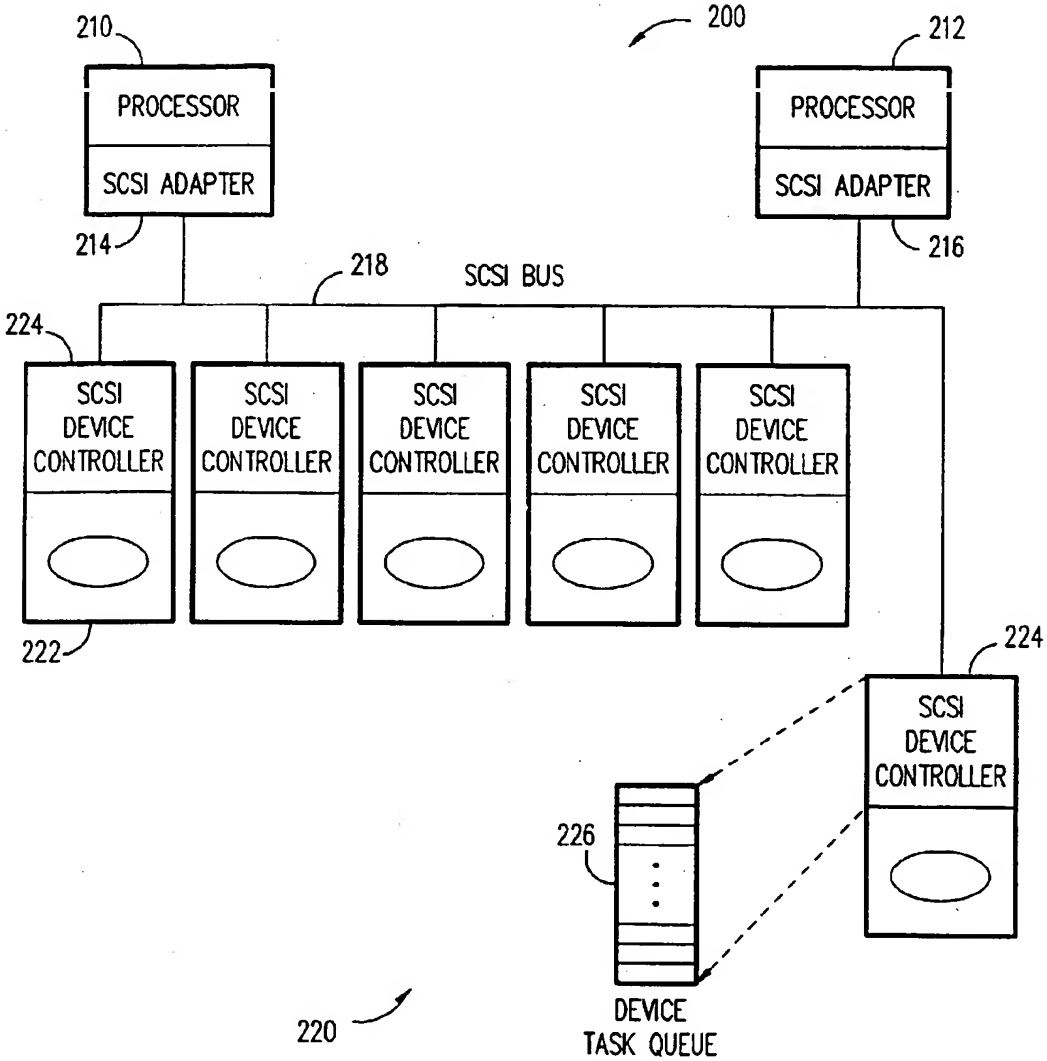

200
210
PROCESSOR
SCSI ADAPTER
214
212
PROCESSOR
SCSI ADAPTER
216
218
SCSI BUS
224
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
222
224
SCSI DEVICE CONTROLLER
226
DEVICE TASK QUEUE
220

FIG. 3

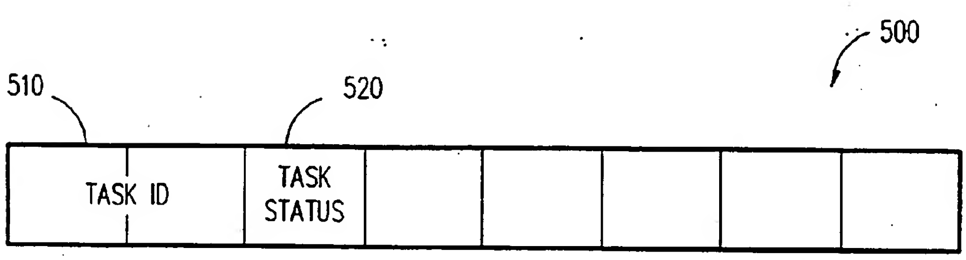
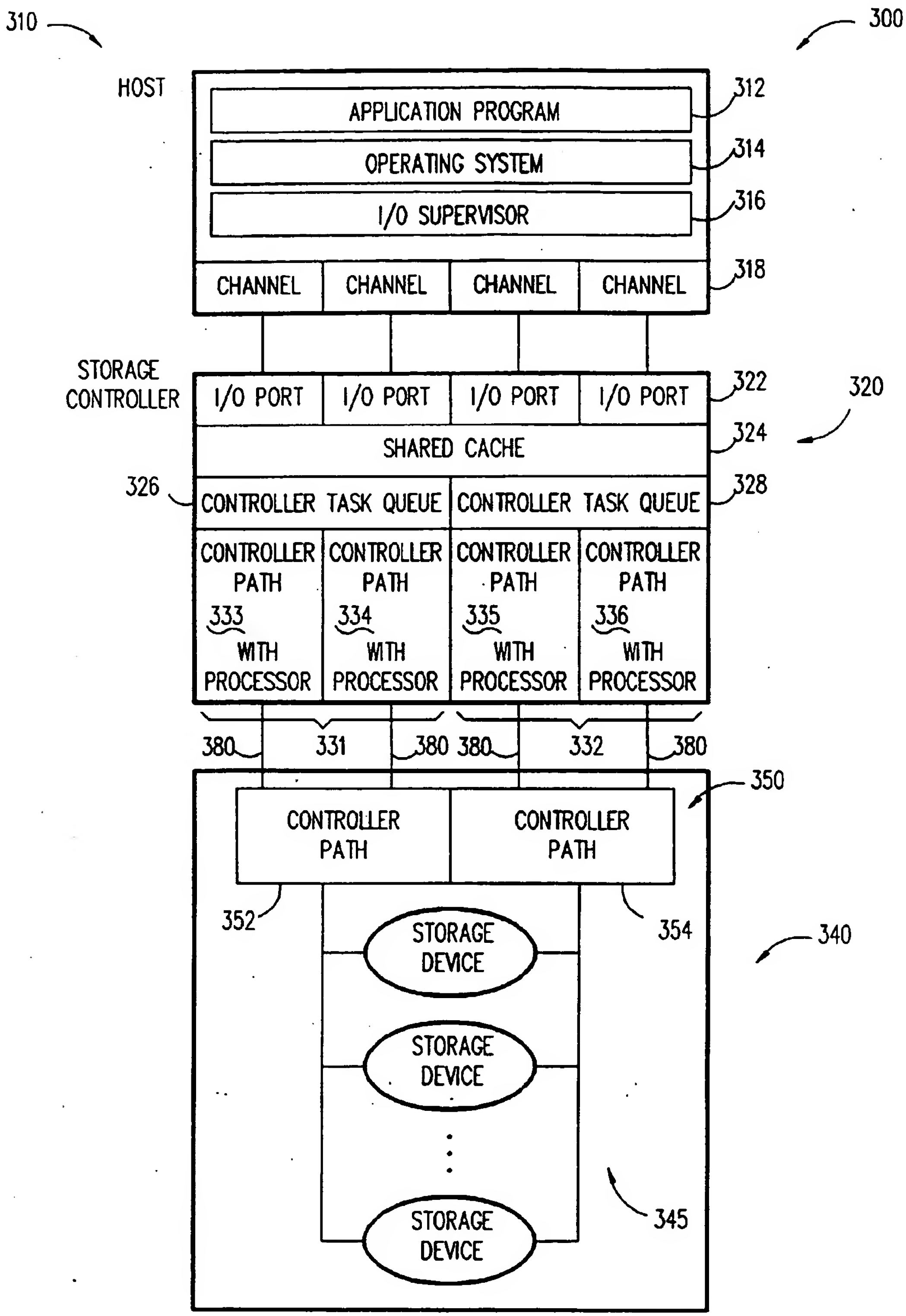

500
510
520
TASK ID
TASK STATUS

FIG. 6



310
300
HOST
APPLICATION PROGRAM
312
OPERATING SYSTEM
314
I/O SUPERVISOR
316
CHANNEL
CHANNEL
CHANNEL
CHANNEL
318
STORAGE CONTROLLER
I/O PORT
I/O PORT
I/O PORT
I/O PORT
322
320
SHARED CACHE
324
326
CONTROLLER TASK QUEUE
CONTROLLER TASK QUEUE
328
CONTROLLER PATH 333 WITH PROCESSOR
CONTROLLER PATH 334 WITH PROCESSOR
CONTROLLER PATH 335 WITH PROCESSOR
CONTROLLER PATH 336 WITH PROCESSOR
380
331
380
380
332
380
350
CONTROLLER PATH
CONTROLLER PATH
352
354
340
STORAGE DEVICE
STORAGE DEVICE
STORAGE DEVICE
345

FIG. 4

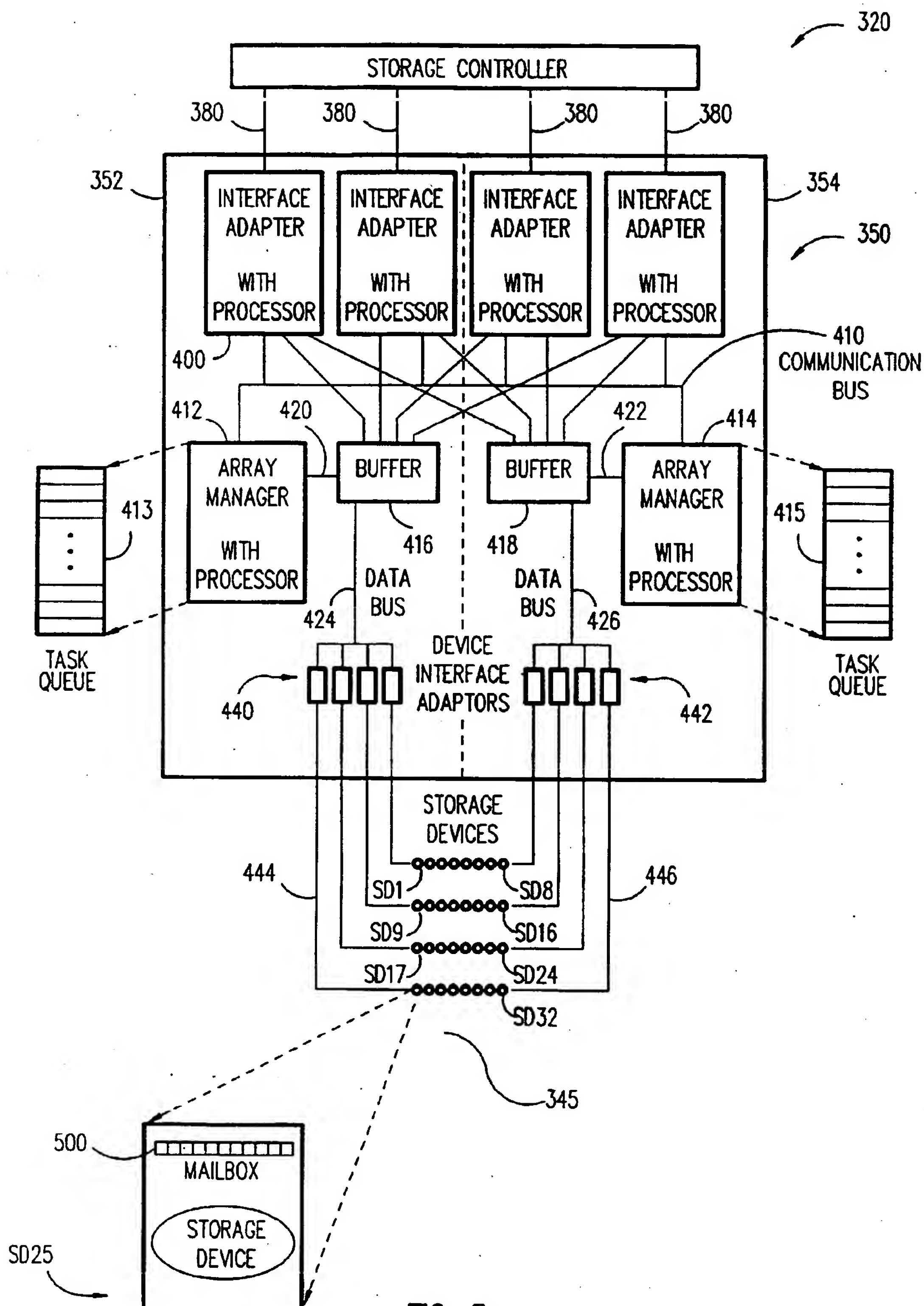

320
STORAGE CONTROLLER
380
380
380
380
352
354
350
INTERFACE ADAPTER WITH PROCESSOR
INTERFACE ADAPTER WITH PROCESSOR
INTERFACE ADAPTER WITH PROCESSOR
INTERFACE ADAPTER WITH PROCESSOR
400
410 COMMUNICATION BUS
412
420
422
414
ARRAY MANAGER WITH PROCESSOR
BUFFER
BUFFER
ARRAY MANAGER WITH PROCESSOR
413
415
416
418
DATA BUS
DATA BUS
424
426
TASK QUEUE
TASK QUEUE
DEVICE INTERFACE ADAPTORS
440
442
STORAGE DEVICES
444
446
SD1
SD8
SD9
SD16
SD17
SD24
SD32
345
500
MAILBOX
STORAGE DEVICE
SD25

FIG. 5

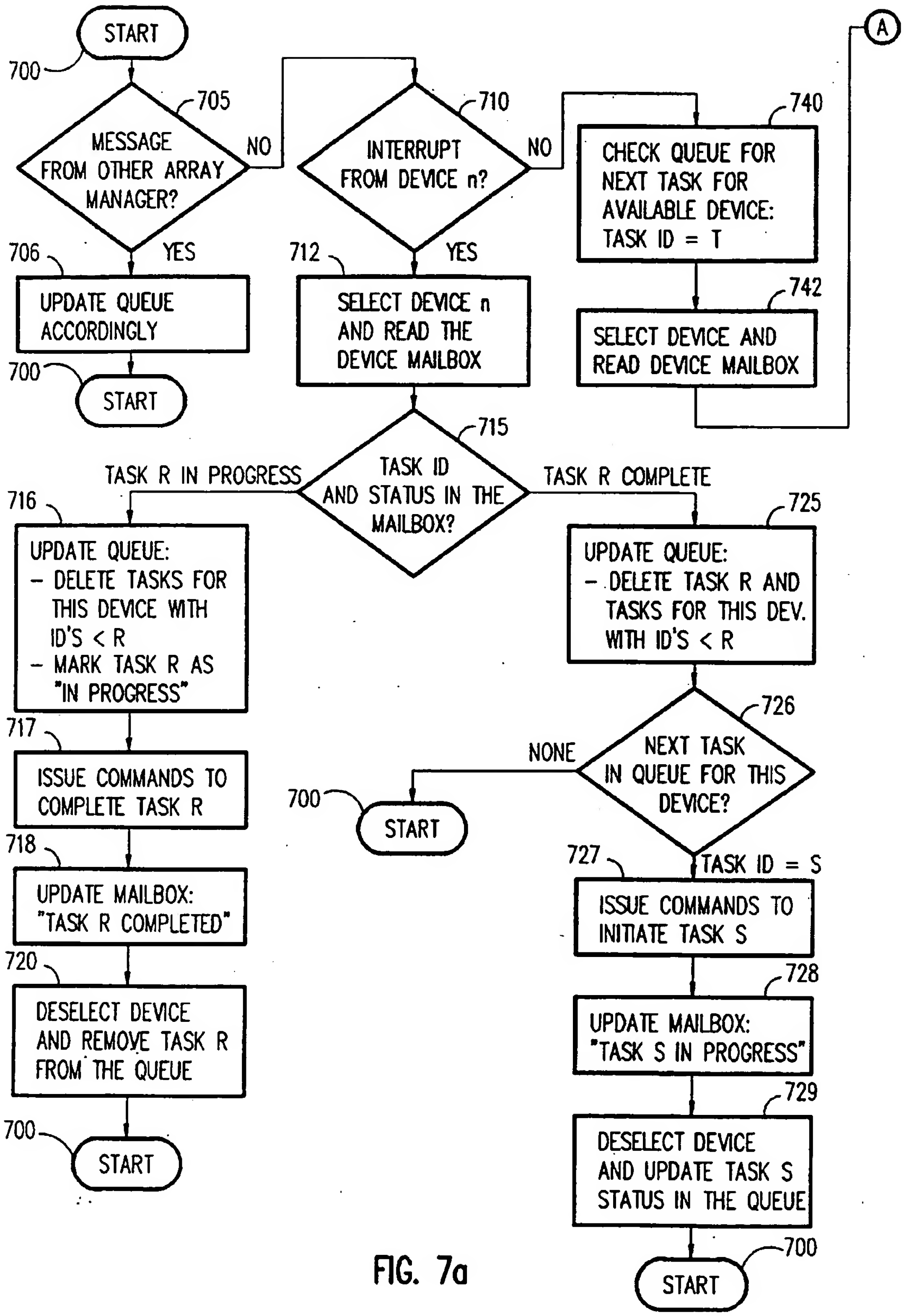

START
700
705
MESSAGE FROM OTHER ARRAY MANAGER?
NO
YES
706
UPDATE QUEUE ACCORDINGLY
700
START
710
INTERRUPT FROM DEVICE n?
NO
YES
712
SELECT DEVICE n AND READ THE DEVICE MAILBOX
740
CHECK QUEUE FOR NEXT TASK FOR AVAILABLE DEVICE: TASK ID = T
742
SELECT DEVICE AND READ DEVICE MAILBOX
A
715
TASK ID AND STATUS IN THE MAILBOX?
TASK R IN PROGRESS
TASK R COMPLETE
716
UPDATE QUEUE:
– DELETE TASKS FOR THIS DEVICE WITH ID'S < R
– MARK TASK R AS "IN PROGRESS"
717
ISSUE COMMANDS TO COMPLETE TASK R
718
UPDATE MAILBOX: "TASK R COMPLETED"
720
DESELECT DEVICE AND REMOVE TASK R FROM THE QUEUE
700
START
725
UPDATE QUEUE:
– DELETE TASK R AND TASKS FOR THIS DEV. WITH ID'S < R
726
NEXT TASK IN QUEUE FOR THIS DEVICE?
NONE
700
START
727
TASK ID = S
ISSUE COMMANDS TO INITIATE TASK S
728
UPDATE MAILBOX: "TASK S IN PROGRESS"
729
DESELECT DEVICE AND UPDATE TASK S STATUS IN THE QUEUE
700
START

FIG. 7a

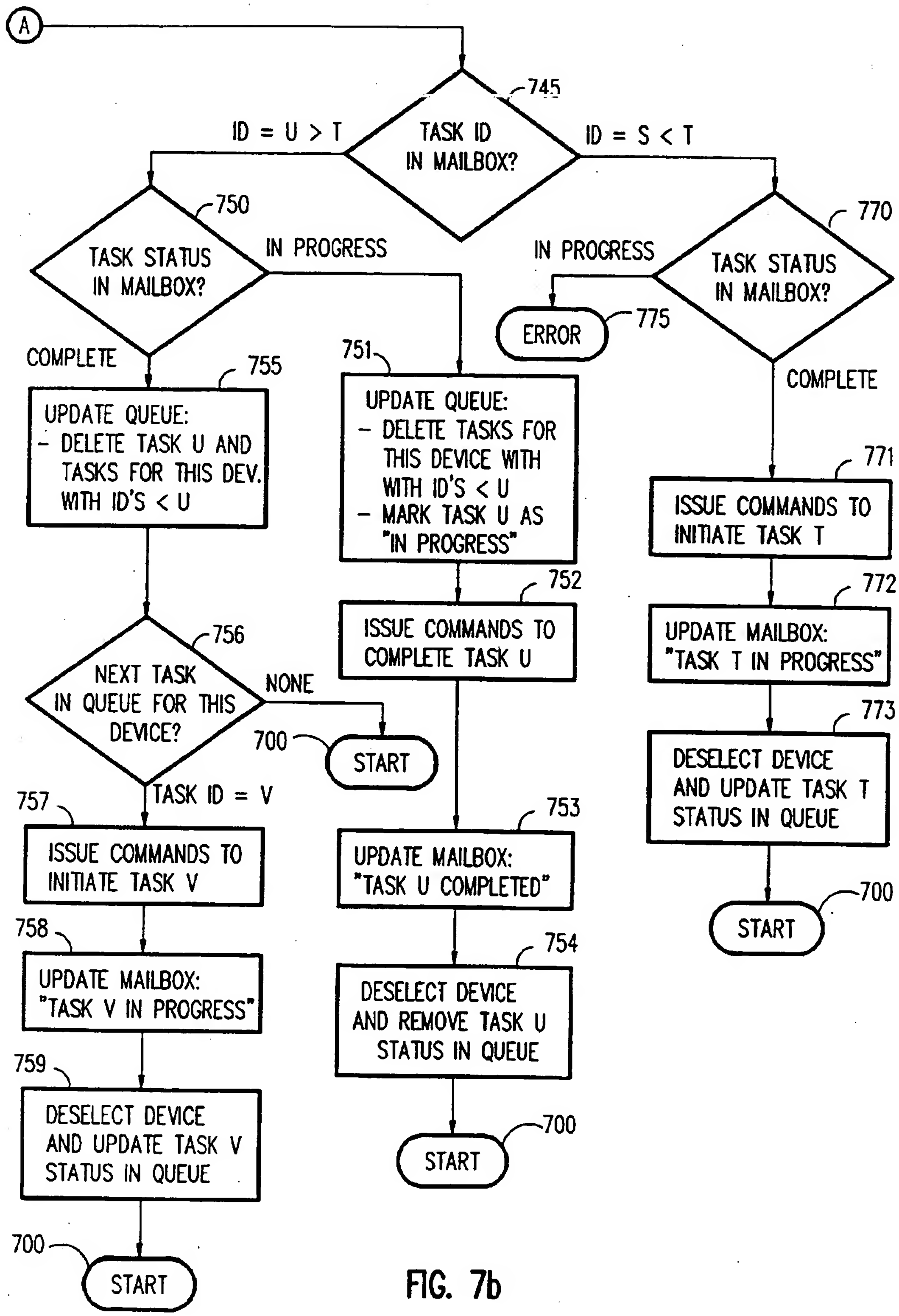

A
745
TASK ID
IN MAILBOX?
ID = U > T
ID = S < T
750
TASK STATUS
IN MAILBOX?
IN PROGRESS
COMPLETE
755
UPDATE QUEUE:
- DELETE TASK U AND
TASKS FOR THIS DEV.
WITH ID'S < U
751
UPDATE QUEUE:
- DELETE TASKS FOR
THIS DEVICE WITH
WITH ID'S < U
- MARK TASK U AS
"IN PROGRESS"
752
ISSUE COMMANDS TO
COMPLETE TASK U
756
NEXT TASK
IN QUEUE FOR THIS
DEVICE?
NONE
700
START
TASK ID = V
757
ISSUE COMMANDS TO
INITIATE TASK V
758
UPDATE MAILBOX:
"TASK V IN PROGRESS"
759
DESELECT DEVICE
AND UPDATE TASK V
STATUS IN QUEUE
700
START
753
UPDATE MAILBOX:
"TASK U COMPLETED"
754
DESELECT DEVICE
AND REMOVE TASK U
STATUS IN QUEUE
700
START
770
TASK STATUS
IN MAILBOX?
IN PROGRESS
ERROR
775
COMPLETE
771
ISSUE COMMANDS TO
INITIATE TASK T
772
UPDATE MAILBOX:
"TASK T IN PROGRESS"
773
DESELECT DEVICE
AND UPDATE TASK T
STATUS IN QUEUE
700
START

FIG. 7b

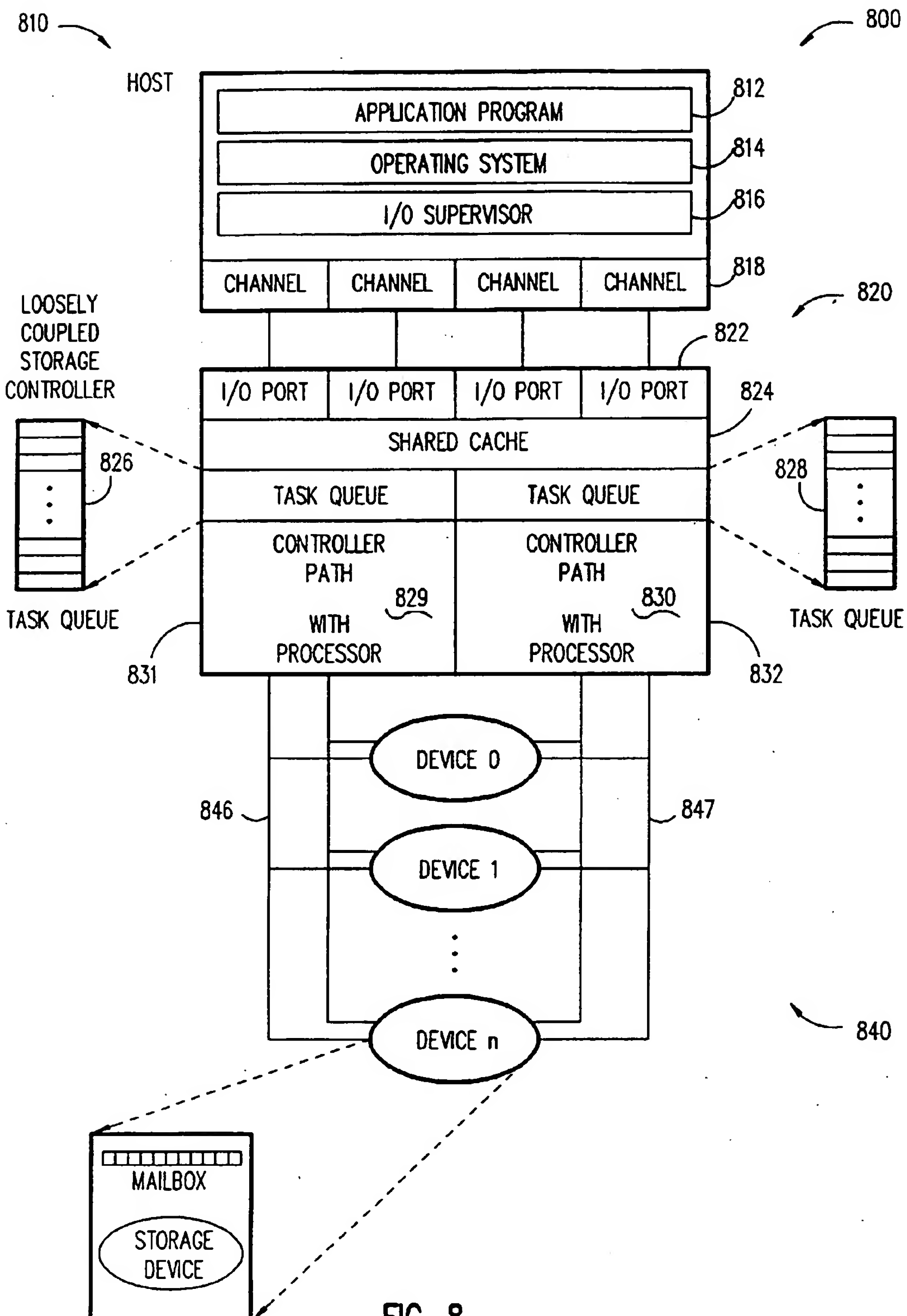

810
800
HOST
APPLICATION PROGRAM
OPERATING SYSTEM
I/O SUPERVISOR
812
814
816
CHANNEL
CHANNEL
CHANNEL
CHANNEL
818
820
822
LOOSELY
COUPLED
STORAGE
CONTROLLER
I/O PORT
I/O PORT
I/O PORT
I/O PORT
824
SHARED CACHE
826
828
TASK QUEUE
TASK QUEUE
CONTROLLER
PATH
WITH
PROCESSOR
CONTROLLER
PATH
WITH
PROCESSOR
829
830
TASK QUEUE
TASK QUEUE
831
832
DEVICE 0
846
847
DEVICE 1
DEVICE n
840
MAILBOX
STORAGE
DEVICE

FIG. 8

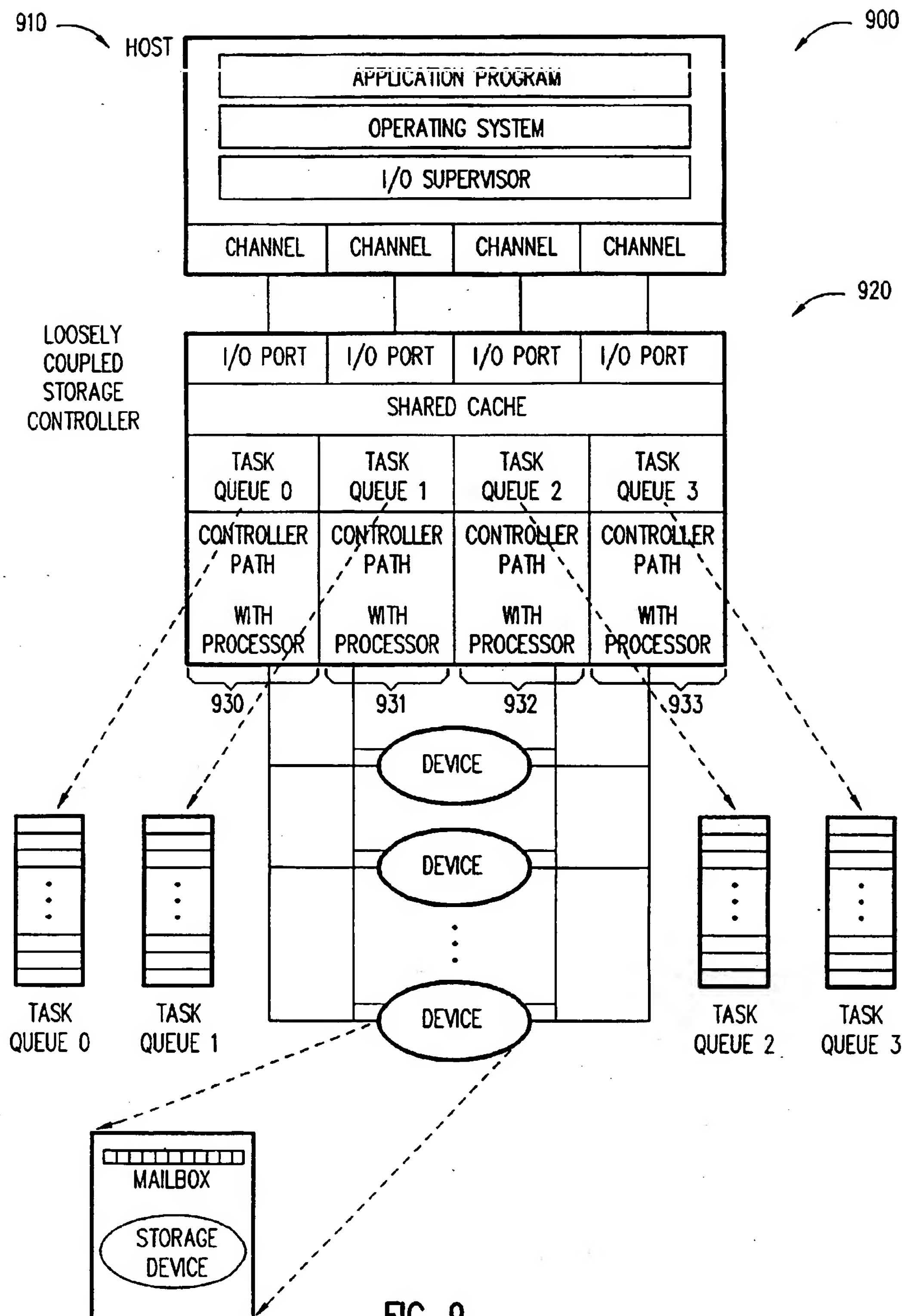

910
HOST
900
APPLICATION PROGRAM
OPERATING SYSTEM
I/O SUPERVISOR
CHANNEL
CHANNEL
CHANNEL
CHANNEL
920
LOOSELY COUPLED STORAGE CONTROLLER
I/O PORT
I/O PORT
I/O PORT
I/O PORT
SHARED CACHE
TASK QUEUE 0
TASK QUEUE 1
TASK QUEUE 2
TASK QUEUE 3
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
930
931
932
933
DEVICE
DEVICE
DEVICE
TASK QUEUE 0
TASK QUEUE 1
TASK QUEUE 2
TASK QUEUE 3
MAILBOX
STORAGE DEVICE

FIG. 9

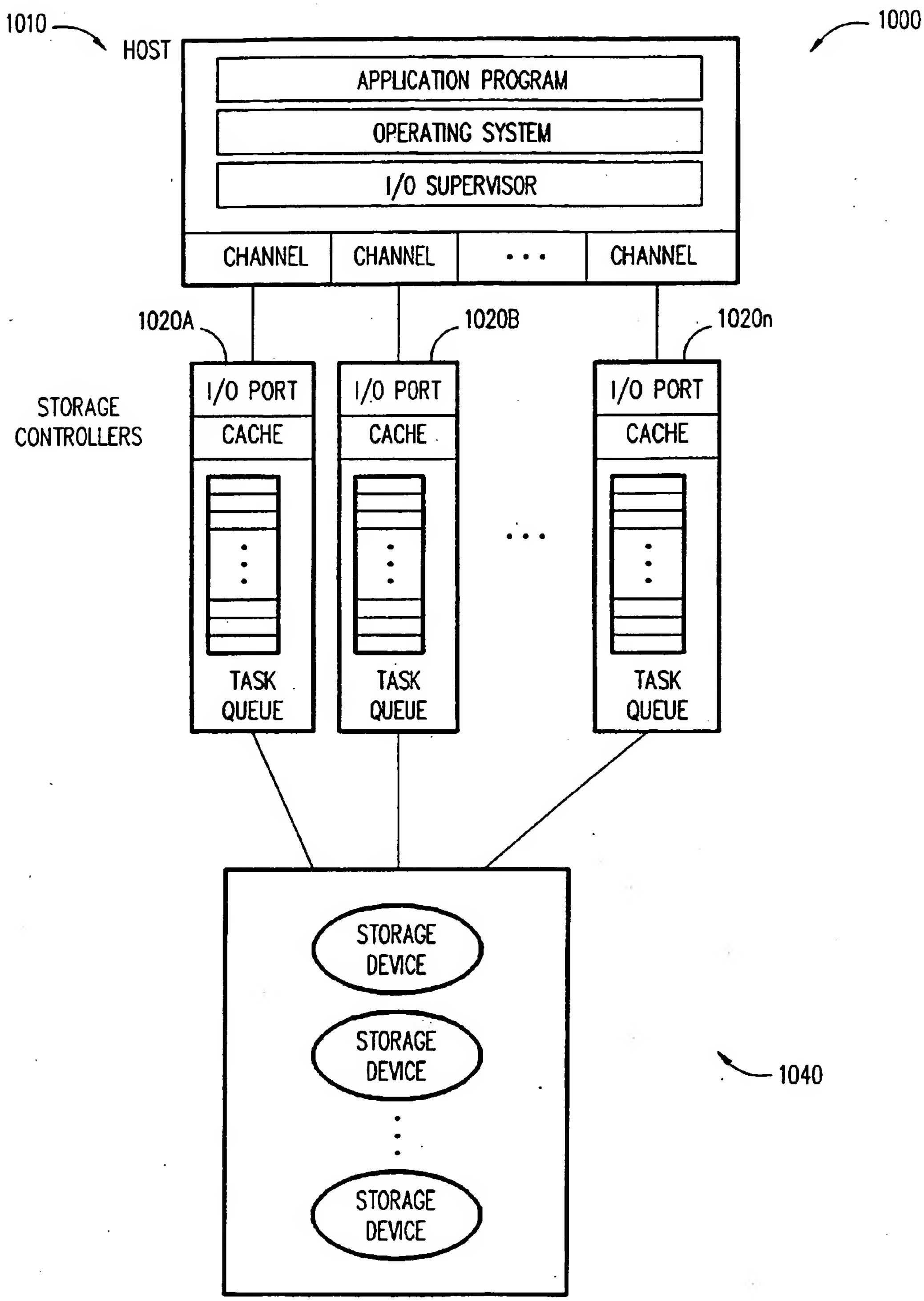

1010
HOST
1000
APPLICATION PROGRAM
OPERATING SYSTEM
I/O SUPERVISOR
CHANNEL
CHANNEL
• • •
CHANNEL
1020A
1020B
1020n
STORAGE CONTROLLERS
I/O PORT
CACHE
TASK QUEUE
I/O PORT
CACHE
TASK QUEUE
I/O PORT
CACHE
TASK QUEUE
STORAGE DEVICE
STORAGE DEVICE
STORAGE DEVICE
1040

FIG. 10